# 第二十六节 分布式配置

## 内部配置（Internal Configuration）

java.util.concurrent.ForkJoinPool#makeCommonPool()

既有内部配置，又有外部配置

## Spring 配置

### 配置源 - PropertySources

#### 属性处理 - PropertyResolver

- 属性存储
- 属性占位符处理

# 配置类型转换 - ConversionService

# Spring Cloud 配置基础

# Spring Cloud Config 架构设计

Spring Cloud Config 是可选的重要组件

## Spring Cloud Config Client

配置信息 - Environment 抽象中的 PropertySources，比如：

```yaml
spring:
# 配置 Spring Cloud Config Server
  ## 配置直连模式
  config:
    import:
'optional:configserver:http://127.0.0.1:8888'
  ## 配置服务发现模式
  cloud:
    config:
      discovery:
        enabled: true
        serviceId: config-server
```

通过连接配置服务器，获取新的远程配置，即新的 PropertySource

## 底层实现 - ConfigServicePropertySourceLocator

- 基于 Spring Cloud Commons 抽象中的 Bootstrap
  - PropertySourceLocator 实现类 - org.springframework.cloud.config.client.ConfigServicePropertySourceLocator
  - 存在 `META-INF/spring.factories` 的配置：

    ```
    org.springframework.cloud.bootstrap.BootstrapConfiguration=\
    org.springframework.cloud.config.client.ConfigServicePropertySourceLocator
    ```

# Spring Cloud Config Server

## 激活配置服务器逻辑

通过 `@EnableConfigServer`

- `@Import` -> ConfigServerConfiguration
  - @BeanConfigServerConfiguration.Marker 被初始化
    - 作为 ConfigServerAutoConfiguration 的判断条件
      - `@ConditionalOnBean(ConfigServerConfiguration.Marker.class)`

## 配置服务器自动装配类 - ConfigServerAutoConfiguration

### 导入配置

- Environment 仓库
  - EnvironmentRepositoryConfiguration
  - CompositeConfiguration （组合多个 EnvironmentRepository）
- ResourceRepositoryConfiguration
- Config Server Web MVC 相关

    - ConfigServerMvcConfiguration
        - EnvironmentController
- 加密相关
    - ConfigServerEncryptionConfiguration
    - ResourceEncryptorConfiguration

## 配置服务器 MVC Controller - EnvironmentController

### 映射路径

- `/{name}/{profiles:.*[^-].*}`

## Web Endpoint

# 连接模式

## 地址直连模式

# Spring Cloud Config 核心 API - PropertySourceLocator

## 内建实现实现

### Spring Cloud Config Client 实现 - ConfigServicePropertySourceLocator

### 特定客户端实现

- Zookeeper - org.springframework.cloud.zookeeper.config.ZookeeperPropertySourceLocator
- Consul - org.springframework.cloud.consul.config.ConsulPropertySourceLocator
- Nacos - com.alibaba.cloud.nacos.client.NacosPropertySourceLocator
- Kubernetes(ConfigMap) - org.springframework.cloud.kubernetes.fabric8.config.Fabric8ConfigMapPropertySourceLocator

## 自定义 Bootstrap PropertySource

参考文档：https://docs.spring.io/spring-cloud-commons/docs/3.0.3/reference/html/#customizing-bootstrap-property-sources

实现步骤：

- 实现 `PropertySourceLocator` 接口
- 配置实现 - `META-INF/spring.factories` 文件
  - 配置
    - `org.springframework.cloud.bootstrap.BootstrapConfiguration=${PropertySourceLocator 实现类}`
- API 实现 - 定义 `PropertySourceLocator` @Bean

## Bootstrap PropertySource 原理

参考 org.springframework.cloud.bootstrap.config.PropertySourceBootstrapConfiguration 实现，

通过内建或者自定义 PropertySourceLocator Bean（集合）来补充 Bootstrap ApplicationContext 中的 PropertySources

```
    @Autowired(required = false)
    private List<PropertySourceLocator>
propertySourceLocators = new ArrayList<>();
```

```
@Override
    public void
initialize(ConfigurableApplicationContext
applicationContext) {
        List<PropertySource<?>> composite = new
ArrayList<>();

AnnotationAwareOrderComparator.sort(this.proper
tySourceLocators);
        boolean empty = true;
        ConfigurableEnvironment environment =
applicationContext.getEnvironment();
        for (PropertySourceLocator locator :
this.propertySourceLocators) {
            Collection<PropertySource<?>>
source = locator.locateCollection(environment);
            if (source == null || source.size()
== 0) {
                continue;
            }
            List<PropertySource<?>> sourceList
= new ArrayList<>();
            for (PropertySource<?> p : source)
{
                if (p instanceof
EnumerablePropertySource) {
                    EnumerablePropertySource<?>
enumerable = (EnumerablePropertySource<?>) p;
                    sourceList.add(new
BootstrapPropertySource<>(enumerable));
                }
                else {
```

```
                    sourceList.add(new
SimpleBootstrapPropertySource(p));
                }
            }
            logger.info("Located property
source: " + sourceList);
            composite.addAll(sourceList);
            empty = false;
        }
        if (!empty) {
            MutablePropertySources
propertySources =
environment.getPropertySources();
            String logConfig =
environment.resolvePlaceholders("${logging.conf
ig:}");
            LogFile logFile =
LogFile.get(environment);
            for (PropertySource<?> p :
environment.getPropertySources()) {
                if
(p.getName().startsWith(BOOTSTRAP_PROPERTY_SOUR
CE_NAME)) {

propertySources.remove(p.getName());
                }
            }

insertPropertySources(propertySources,
composite);

reinitializeLoggingSystem(environment,
logConfig, logFile);
```

```
            setLogLevels(applicationContext,
environment);

handleIncludedProfiles(environment);
        }
    }
```

# 相关议题

## 配置中心和注册中心

注册中心，读多写多，不强依赖于数据持久化，更偏好于高可用，最终一致性即可

> 服务消费客户端本地缓存一分订阅服务的列表数据

配置中心，读多写少，推荐数据持久化。

### 如何配置 Spring Cloud 依赖

- 如果使用 Spring Cloud Alibaba 的话，那么可以参考：

https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E#%E6%AF%95%E4%B8%9A%E7%89%88%E6%9C%AC%E4%BE%9D%E8%B5%96%E5%85%B3%E7%B3%BB%E6%8E%A8%E8%8D%90%E4%BD%BF%E7%94%A8

| Spring Cloud Version | Spring Cloud Alibaba Version | Spring Boot Version |
| --- | --- | --- |
| Spring Cloud 2020.0.0 | 2021.1 | 2.4.2 |
| Spring Cloud Hoxton.SR8 | 2.2.5.RELEASE | 2.3.2.RELEASE |
| Spring Cloud Greenwich.SR6 | 2.1.4.RELEASE | 2.1.13.RELEASE |

- 如果仅使用了 Spring Cloud 的话，那么可以参考：

通过 https://start.spring.io/ 配置对应版本，选择 "EXPLORE" 后，会弹出 Maven 版本信息，比如：

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/P
OM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
```

```
    <version>2.5.0</version>
    <relativePath/> <!-- lookup parent from
repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring
Boot</description>
  <properties>
    <java.version>11</java.version>
    <spring-cloud.version>2020.0.3-
SNAPSHOT</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>

 <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-
server</artifactId>
    </dependency>
  ...
```

## @EnableAutoConfiguration 与 @BootstrapConfiguration

@EnableAutoConfiguration 隶属于 Spring Boot，为 Spring Boot ApplicationContext 提供自动装配 Configuration Class

@BootstrapConfiguration 隶属于 Spring Cloud（Spring Cloud Commons），为 Spring Cloud Bootstrap ApplicationContext（Spring Boot ApplicationContext 的 Parent ApplicationContext）配置 Bean

# Spring Cloud Bootstrap ApplicationContext 与 Spring Boot ApplicationContext 的关系

Spring Cloud 应用是一个 Spring Boot 应用，利用 Spring Boot 生命周期，创建 Bootstrap ApplicationContext，并将其作为 Spring Boot ApplicationContext 的 Parent ApplicationContext，同时，优先启动。

Spring Boot ApplicationContextion 在准备过程中启动了一个 Bootstrap ApplicationContext

## SpringApplication 生命周期

总入口 run() 方法

- 准备 Environment - org.springframework.boot.SpringApplication#prepareEnvironment

    - 成员 SpringApplicationRunListener 有一个默认实现 - org.springframework.boot.context.event.EventPublishingRunListener#environmentPrepared
        - 发送 org.springframework.boot.context.event.ApplicationEnvironmentPreparedEvent
- 创建 ApplicationContext - org.springframework.boot.SpringApplication#createApplicationContext
- 准备（初始化）ApplicationContext - org.springframework.boot.SpringApplication#prepareContext
    - org.springframework.boot.SpringApplicationRunListeners
        - 成员 SpringApplicationRunListener 有一个默认实现 - org.springframework.boot.context.event.EventPublishingRunListener#contextPrepared
            - 发送 org.springframework.boot.context.event.ApplicationContextInitializedEvent 事件
- 启动 ApplicationContext - org.springframework.boot.SpringApplication#refreshContext

## Bootstrap ApplicationContext 创建并启动

该功能隶属于 Spring Cloud Commons 模块，利用了 Spring Boot SpringApplication 生命周期来扩展实现。

Spring Cloud Commons 模块提供了一个 ApplicationListener 实现，去监听 ApplicationEnvironmentPreparedEvent 事件，即：org.springframework.cloud.bootstrap.BootstrapApplicationListener

```
    @Override
    public void
onApplicationEvent(ApplicationEnvironmentPrepar
edEvent event) {
        ConfigurableEnvironment environment =
event.getEnvironment();
        if (!bootstrapEnabled(environment) &&
!useLegacyProcessing(environment)) {
            return;
        }
        // don't listen to events in a
bootstrap context
        if
(environment.getPropertySources().contains(BOOT
STRAP_PROPERTY_SOURCE_NAME)) {
            return;
        }
        ConfigurableApplicationContext context
= null;
        String configName =
environment.resolvePlaceholders("${spring.cloud
.bootstrap.name:bootstrap}");
```

```
        for (ApplicationContextInitializer<?>
initializer :
event.getSpringApplication().getInitializers())
{
            if (initializer instanceof
ParentContextApplicationContextInitializer) {
                context =
findBootstrapContext((ParentContextApplicationC
ontextInitializer) initializer, configName);
            }
        }
        if (context == null) {
            context =
bootstrapServiceContext(environment,
event.getSpringApplication(), configName);

event.getSpringApplication().addListeners(new
CloseContextOnFailureApplicationListener(contex
t));
        }

        apply(context,
event.getSpringApplication(), environment);
    }
```

其中 SpringApplicationBuilder#build 生成 SpringApplication 对象，SpringApplication#run 方法会创建自己的 ApplicationContext，即 bootstrap ApplicationContext。

总之，Spring Cloud Bootstrap ApplicationContext 是基于 Spring Boot SpringApplication 实现

Spring Boot Spring Application#run()

- 准备 Environment
  - 发送 ApplicationEnvironmentPreparedEvent
    - 被 BootstrapApplicationListener 监听事件
      - 创建 SpringApplication
        - 调用 SpringApplication#run()
          - 创建 Bootstrap ApplicationContext

## ApplicationContext Parent 关联

```
    @Override
    public void setParent(@Nullable
ApplicationContext parent) {
        this.parent = parent;
        if (parent != null) {
            Environment parentEnvironment =
parent.getEnvironment();
            if (parentEnvironment instanceof
ConfigurableEnvironment) {

getEnvironment().merge((ConfigurableEnvironment
) parentEnvironment);
            }
        }
    }
```

# Environment、ApplicationContext 与 SpringApplication 的关系

- 一个 SpringApplication 创建并且关联 一个 ApplicationContext
- 一个 ApplicationContext 管理着一个 Environment 对象
- 一个 Environment 对象关联了一个 PropertySources
- 一个 PropertySources 关联了一个或多个 PropertySource

# Bootstrap ApplicationContext 关联 Bootstrap Environment 初始化过程

- 创建阶段：Bootstrap ApplicationContext 在创建过程中会创建一个 StandardEnvironment 对象，即 Bootstrap Environment

```
StandardEnvironment bootstrapEnvironment =
new StandardEnvironment();
        MutablePropertySources
bootstrapProperties =
bootstrapEnvironment.getPropertySources();
```

- 初始化阶段：根据 "spring.cloud.bootstrap.*" 配置信息获取 Bootstrap 初始化 PropertySource

- 扩展阶段：通过 org.springframework.cloud.bootstrap.config.PropertySourceBootstrapConfiguration
    - 收集 PropertySourceLocator Bean 集合，并将 locate PropertySource 添加到 Bootstrap Environment 中，参考：org.springframework.cloud.bootstrap.config.PropertySourceBootstrapConfiguration#initialize

## BootstrapApplicationListener会有早期事件的问题吗?

答：不会，因为 org.springframework.boot.context.event.EventPublishingRunListener 关联的 org.springframework.context.event.SimpleApplicationEventMulticaster 是独立于 ApplicationContext 实现。

## Spring Cloud Config Client 实现与 Spring Cloud Config 特定客户端实现的区别

如果配置客户端使用 Spring Cloud Config Client 的方式（标准 REST API 方式），若要添加新配置存储的实现，需要实现 Spring Cloud Config Server，比如 Spring Cloud Alibaba Nacos Config Server 实现 。优点：无需升级依赖，只需要调

整配置即可。（少数）比如，将配置从 Git 迁移到 Nacos。

反之，如果使用 Spring Cloud Config 特定客户端实现，需要利用 PropertySourceLocator 来提供特定配置存储基础设施实现，比如 Spring Cloud Alibaba Nacos Config 实现 。优点：性能和效率会比较占优势。（多数），比如，将 Zookeeper 作为配置管理替换为 Nacos 的实现。

# 作业

基于文件系统为 Spring Cloud 提供 PropertySourceLocator 实现

- 配置文件命名规则（META-INF/config/default.properties 或者 META-INF/config/default.yaml）